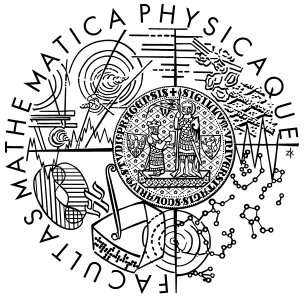
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Tomáš Mikuš

Kompresný program

Katedra softwarového inženýrství

Vedúci bakalárskej práce: Mgr. Mikuláš Patočka

Študijný program: Správa počítačových systémů

2007

Ďakujem Mikulášovi Patočkovi za cenné rady a pripomienky k programu, ktoré výrazne prispeli k jeho zlepšeniu.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičaním práce a jej zverejňovaním.

V Prahe dňa 30.5.2007 Tomáš Mikuš

Obsah

Názov práce: Kompresný program

Autor: Tomáš Mikuš

Katedra: Katedra softwarového inženýrství

Vedúci bakalárskej práce: Mgr. Mikuláš Patočka

e-mail vedúceho: mikulas.patocka@mff.cuni.cz

Abstrakt: V predloženej práci študujeme možnosti kompresie dát. Pokúsime sa popísať a neskôr implementovať kompresný algoritmus porovnateľnej rýchlosti a efektivity ako algoritmus, ktorý je využívaný programom bzip2. Tento algoritmus bude základom programu bežiaceho pod konzolou operačného systému Linux, ktorý bude spĺňať nasledujúce vlastnosti: Bude poskytovať primerané grafické užívateľské prostredie na pohodlnú a jednoduchú prácu s archívom. Bude implementovať funkcie, medzi ktoré bude patriť vytvorenie archívu zo zadaných súborov a adresárov, pridanie ďalších súborov alebo ich odoberanie z archívu a samozrejme spätné obnovenie súborov a adresárov. Ďalšou funkciou bude možnosť prechádzania adresárového stromu obsiahnutého v archíve.

Kľúčové slová: kompresia, bwt, aritmetické kódovanie

Title: Compression program

Author:Tomáš Mikuš

Department: Department of Software Engineering

Supervisor: Mgr. Mikuláš Patočka

Supervisor's e-mail address: mikulas.patocka@mff.cuni.cz

Abstract: In the present work we study posibilities of data comression. We will try to describe and later implement compression algorithm with speed and efficiency comparable with algorithm that is used in bzip2 program. This algorithm will be used in program running under console of Linux operating system. This program will meet following conditions: It will have graphic user interface for easy work with archiv allowing user to create archiv file from requested files and directories, add, and delete other files from archiv.and also decompress the content of archiv. User will be able to browse directory tree stored in archiv.

Keywords: compression, bwt, arithmetic coding

Členenie kapitol

1. kapitola tejto práce oboznamuje čitateľa s cieľmi a úlohami, o ktorých splnenie sa táto bakalárska práca s priloženým program snaží.

2. kapitola čitateľovi priblíži problematiku kompresie dát a vysvetlí, ako teoreticky použité algoritmy fungujú.

3. kapitola popisuje presnú štruktúru a spôsob práce programu. Zároveň je tu opísaný formát archívu a spôsob editovania.

4. kapitola sa podrobnejšie zaoberá princípom práce použitých algoritmov a popisom ich implementácie.

5. kapitola porovnáva výkon kompresného algoritmu, teda jeho rýchlosť a kompresný pomer s programom bzip2.

6. kapitola obsahuje návod na použitie a popis užívateľského prostredia.

7. kapitola zhŕňa nakoľko sa podarilo splniť stanovené úlohy a ciele.

Súčasťou práce je priložený CD nosič s programom.

Kapitola 1

Úvod

1.1 Úvod

Cieľom tejto bakalárskej práce je vytvorenie aplikácie, ktorá bude určená na kompresiu súborov a adresárov. Má užívateľovi poskytnúť grafické prostredie, v ktorom sa súbory a adresáre budú dať jednoducho komprimovať. Ďalšou funkciou, ktorá má byť v programe implementovaná je možnosť prechádzať adresárovým stromom uloženým v archíve a jeho následná úprava, teda pridávanie a odoberanie ďalších súborov a adresárov.

Aplikácia by mala využívať vlastnú implementáciu kompresného algoritmu, ktorý by sa rýchlosťou a efektivitou kompresie mal približovať algoritmu používaného v programe bzip2.

1.2 Motivácia

Motiváciou k napísaniu tejto práce a vytvoreniu už spomínaného programu je uľahčenie práce užívateľom spojenej s komprimovaním. V prostredí konzoly operačného systému na báze Unix je kompresia dát takmer vždy spojená s vypísaním minimálne dvoch príkazov a zistením tých správnych parametrov. Pri potrebe doplnenia, zmeny alebo len zistenia obsahu archívu je to ešte zložitejšie. Program by mal pomôcť najmä začínajúcim a občasným používateľom.

Kapitola 2

Kompresia dát

2.1 Čo je to kompresia

Cieľom kompresie je zmenšenie objemu. V prípade počítačov ide o proces zmenšovania objemu dát. Dôvodom na kompresiu je zvyčajne snaha o efektívnejšie využitie pamäťových medií, alebo zvýšenie prenosovej kapacity nejakého informačného kanálu. Kompresia v podstate odstraňuje nadbytočné časti údajov, aby zostal zachovaný ich obsah, alebo sa zmenil len v prijateľnej miere.

2.2 História kompresie dát

Za počiatok kompresie dát sa dá považovať známa Morseova abeceda z roku 1838. Tá na zmenšenie objemu prenášaných údajov cez telegraf (množstvo bodiek a čiarok) využíva rôznu dĺžku kódov pre jednotlivé písmenká podľa toho, ako často sa vyskytujú v bežnom anglickom texte. Preto pre často sa vyskytujúce E je použitá jedna bodka a pre málo sa vyskytujúce H až štyri.

2.3 Základné delenie metód

Hlavným delením metód na kompresiu dát je delenie na stratovú a bezstratovú kompresiu. Rozdiel spočíva v tom, že pri bezstratovej kompresii sa požaduje, aby údaje po dekompresii boli 100% identické s údajmi pred komprimovaním. Táto podmienka musí byť splnená najmä pri kompresii textu a iných bežných súborov. Pri stratovej kompresii je prípustná istá strata údajov. Využíva sa to pri kompresii audia, videa a obrázkov, kde isté zhoršenie kvality (strata informácie) je pre bežného užívateľa nepozorovateľné. Umožňuje to dosiahnuť omnoho väčšiu kompresiu ako pri bezstratovej, zavisí to ale od toho aké zhoršenie kvality je prípustné.

2.4 Run-length kódovanie

Run-length kódovanie je jedna z najjednoduchších metód kompresie a zaraďuje sa do skupiny bezstratových metód. Často sa označuje skratkou RLE (Run-length encoding). Je založená na nahrádzaní postupností rovnakých znakov (behov) ich dĺžkami. Využíva sa napríklad aj vo formáte JPEG na zakódovanie postupností núl.

Existuje veľa rôznych variant ako behy zakódovať. Bližšie si na príklade (obr. 2.1) ukážeme tie najzákladnejšie.

AAAAAAAAAABCCDDDDDDDD

obrázok 2.1: vstupný reťazec pre RLE

Je potrebné, aby sa dalo odlíšiť, čo je len znak a čo je zapísaná dĺžka. Používa sa jednoduché pravidlo, že za každým znakom nasleduje počet opakovaní daného znaku. (obr. 2.2) Tým sa ale predĺži zápis behov dĺžky jedna. Je zrejmé, že pracuje s časovou zložitosťou $O(n)$, kde n je dĺžka vstupného reťazca.

A**9**B**0**C**1**D**7**

obrázok 2.2: výstupný reťazec RLE

V niektorých reťazcoch sa môžu vyskytovať behy dĺžky jedna veľmi často, napríklad aj v bežnom texte. Preto sa častejšie používa pravidlo, kde behy dĺžky jedna zostanú nezmenené a počet opakovaní nasleduje až za behmi dĺžky aspoň dva. (obr. 2.3) To síce predlžuje behy dĺžky dva, ale tie by už nemali byť natoľko časté.

AA**8**BCC**0**DD**6**

obrázok 2.3: výstupný reťazec RLE

RLE ako samostatná metóda na kompresiu je nepostačujúca, ale jej jednoduchosť a rýchlosť jej umožňujú byť súčasťou zložitejších algoritmov.

2.5 Move to front transformácia

Move to front transformácia je kódovanie, ktoré má za cieľ zlepšiť efektivitu entropických kódovaní, ktoré následne dáta spracúvajú. Často je označované skratkou MTF. Samo objem dát nijako nezmenšuje ani nezväčšuje. Mení vždy iba znak za znak, tak aby zvýšilo počet výskytov znakov, ktorých hodnota je blízka nule. Prácu algoritmu si ukážeme na príklade (obr. 2.4)

88083582

obrázok 2.4: vstupný reťazec pre MTF, hodnoty znakov

Označme si k veľkosť abecedy a n dĺžku vstupného reťazca. Algoritmus si na začiatku nainicializuje tabuľku znakov T (obr. 2.5) o veľkosti k, tak aby platilo $T[i]=i$.

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

obrázok 2.5: tabuľka znakov MTF, po inicializácii

Načíta znak s, nájde ho v tabuľke T na pozícii, ktorú si označíme i. Vypíše i ako znak a presunie s na začiatok T.

(8, 0, 1, 2, 3, 4, 5, 6, 7, 9)
obrázok 2.6: tabuľka znakov MTF, po spracovaní 1. znaku

(0, 8, 1, 2, 3, 4, 5, 6, 7, 9)
obrázok 2.7: tabuľka znakov MTF, po spracovaní 3. znaku

(5, 3, 8, 0, 1, 2, 4, 6, 7, 9)
obrázok 2.8: tabuľka znakov MTF, po spracovaní 6. znaku

(2, 8, 5, 3, 0, 1, 4, 6, 7, 9
obrázok 2.9: tabuľka znakov MTF, po spracovaní 8. znaku

80114625
obrázok 2.10: výstupný reťazec pre MTF, hodnoty znakov

Po načítaní druhého znaku sa tabuľka nezmenila, pretože znak 8 už bol na jej začiatku a teda ho nebolo treba presúvať. Súčet hodnôt znakov vo vstupnom reťazci sa z hodnoty 42 zmenšil na hodnotu 27, takže algoritmus svoju úlohu splnil. Nájdenie znaku má časovú zložitosť $O(k)$ a presunutie s s posunutím i znakov tiež, takže celý algoritmus beží so zložitosťou $O(k)$.

2.6 Burrows-Wheelerova transformácia

Úlohou Burrows-Wheelerovej transformácie v kompresnom algoritme je vytvorenie behov rovnakých znakov. To do značnej miery zvýši účinnosť metódy RLE, ktorá následne dáta spracúva. Často sa označuje skratkou BWT.

Jej princíp vychádza z predpokladu, že znaky v reťazcoch nie sú usporiadané úplne náhodne. Z vlastností jazyka vyplýva, že niektoré znaky budú častejšie nasledované jednou postupnosťou ako ostatné a teda, keď sa lexikograficky usporiadajú rotácie nejakého reťazca, tak aj znaky, ktoré im predchádzali budú ako tak usporiadané. Podstatné je, sa že vytvoria behy. Vyššie uvedený princíp neplatí len pre text ale aj pre binárne dáta a BWT je teda univerzálne použiteľná metóda.

Na vstup dostane reťazec znakov, ktorých hodnoty nijako nemení. Zmení len ich poradie. Aby bolo možné údaje vrátiť späť do pôvodného stavu je potrebné pridať ku každému bloku informáciu navyše, ale v porovnaní s veľkosťou bloku je jej veľkosť zanedbateľná.

Postup zakódovania si ukážeme na príklade. (obr. 2.11)

BAKALARSKA_PRACA$
obrázok 2.11: vstupný reťazec BWT

Znakom $ je označený koniec bloku a jeho hodnota je menšia ako hodnota, ktoréhokoľvek iného znaku. Pre ilustráciu si môžme predstaviť, že sa vytvoria všetky

rotácie daného bloku. (obr. 2.12) Jednoduchým presúvaním posledného znaku na začiatok reťazca.

BAKALARSKA_PRACA$
$BAKALARSKA_PRACA
A$BAKALARSKA_PRAC
CA$BAKALARSKA_PRA
ACA$BAKALARSKA_PR
RACA$BAKALARSKA_P
PRACA$BAKALARSKA_
_PRACA$BAKALARSKA
A_PRACA$BAKALARSK
KA_PRACA$BAKALARS
SKA_PRACA$BAKALAR
RSKA_PRACA$BAKALA
ARSKA_PRACA$BAKAL
LARSKA_PRACA$BAKA
ALARSKA_PRACA$BAK
KALARSKA_PRACA$BA
AKALARSKA_PRACA$B

obrázok 2.12: rotácie vstupného reťazca BWT

Takto vzniknuté reťazce sa lexikograficky utriedia. (obr. 2.13)

$BAKALARSKA_PRACA
A$BAKALARSKA_PRAC
ACA$BAKALARSKA_PR
AKALARSKA_PRACA$B
ALARSKA_PRACA$BAK
ARSKA_PRACA$BAKAL
A_PRACA$BAKALARSK
BAKALARSKA_PRACA$ ←
CA$BAKALARSKA_PRA
KALARSKA_PRACA$BA
KA_PRACA$BAKALARS
LARSKA_PRACA$BAKA
PRACA$BAKALARSKA_
RACA$BAKALARSKA_P
RSKA_PRACA$BAKALA
SKA_PRACA$BAKALAR
_PRACA$BAKALARSKA

obrázok: 2.13 utriedené rotácie reťazca BWT

Výstupom transformácie sú posledné znaky z utriedených reťazcov. (obr. 2.14) A informácia koľký v poradí bol vstupný reťazec. Bez tejto informácie sme síce schopný spätne z výstupného reťazca vytvoriť všetky rotácie, ale nevieme povedať, ktorá je tá správna. Pre zjednodušenie necháme znak $ na jeho mieste. Pri

implementácii je tento znak odstránený a uloží sa len jeho pozícia, v tomto prípade 7. Čím sa vyhneme problémom s pridávaním 257. znaku.

ACRBKLK$AASA_PARA

obrázok 2.14: výstupný reťazec BWT

Ako vidieť na (obr. 2.14) veľa behov sa vytvoriť nepodarilo. Je to spôsobené veľmi malým vstupným reťazcom. Zvyčajne sa reťazec spracováva po blokoch o veľkosti 100 – 2500 KB.

Inverzná metóda najprv utriedi svoj vstupný blok a bude postupovať nasledujúcimi pravidlami:

1. začneme na znaku $ (pozícia 7) vo vstupnom bloku
2. prejdeme na rovnakú pozíciu v utriedenom bloku. Znak, ktorý sa tam nachádzal, vypíšeme a označíme ho s. Jeho pozíciu v postupnosti zhodných znakov s s si označíme j.
3. prejdeme na pozíciu vo vstupnom bloku, kde sa nachádza j-ty znak v postupnosti zhodných znakov s s a pokračujeme krokom 2.

V našom prípade 7. pozícia → prvé B → 3. pozícia → tretie A → 9. pozícia → prvé K … Po opätovnom návrate na znak $ bude vypísaný celý výstupný blok a teda algoritmus skončí.

ACRBKLK$AASA_PARA

obrázok 2.15: vstupný reťazec inverznej BWT

$AAAAAABCKKLPRRS_

obrázok 2.16: utriedený vstupný reťazec inverznej BWT

Implementácia algoritmu Burrows-Wheelerovej transformácie, tak ako je popísaný, by bola veľmi neefektívna. Pri veľkosti bloku 1MB by matica všetkých rotácií zaberala 1TB. Aj použitie bežných algoritmov na triedenie reťazcov z BWT veľmi použiteľnú metódu na kompresiu nerobí. Riešením je použitie algoritmov na sufixové triedenie.

2.7 Aritmetické kódovanie

Ide o metódu používanú pri bezstratovej kompresii. Je to entropické kódovanie, ale narozdiel od Huffmanovho nenahrádza každý znak jeho kódom, ale celý reťazec zakóduje do jedného jediného desatinného čísla ležiaceho v intervale $\langle 0\,;1)$. [3]

Na začiatku máme pravdepodobnosti výskytov pre jednotlivé znaky (obr. 2.17), či už sú napevno zadané alebo získané z počtu výskytov znakov, ktoré sme určili jedným prechodom cez vstupný reťazec. Interval $\langle 0\,;1)$ rozdelíme medzi znaky tak, že každému zo znakov pridelíme podinterval, ktorý veľkosťou presne zodpovedá pravdepodobnosti jeho výskytu. (obr. 2.18)

A	0,6
B	0,2
C	0,1
D	0,1

obrázok 2.17: pravdepodobnosti výskytov

A	$\langle 0{,}0\,;\,0{,}6)$
B	$\langle 0{,}6\,;\,0{,}8)$
C	$\langle 0{,}8\,;\,0{,}9)$
D	$\langle 0{,}9\,;\,1{,}0)$

obrázok 2.18: pridelené intervaly

Postupujeme podľa nasledujúcich krokov.

1. Nastavíme interval I na $\langle 0\,;\,1)$
2. Načítame znak s, ak už na vstupe nie sú znaky pokračujeme krokom 4.
3. Nech pre znak s máme interval začínajúci v bode D_s dĺžky R_s a pre I začínajúci v bode D dĺžky R, nový interval pre I sa spočíta podľa (2.1) a (2.2), pokračujeme krokom 2.
4. Na výstup dáme číslo s najkratším desatinným rozvojom z intervalu I

$$D = D + D_s * R \tag{2.1}$$

$$R = R * R_s \tag{2.2}$$

Príklad pre vstupný reťazec ACD:

$I = \langle 0\,;\,1)$

načítame A, $I = \langle 0{,}0\,;\,0{,}6)$

načítame C, $I = \langle 0{,}48\,;\,0{,}54)$

načítame D, $I = \langle 0{,}534\,;\,0{,}54)$

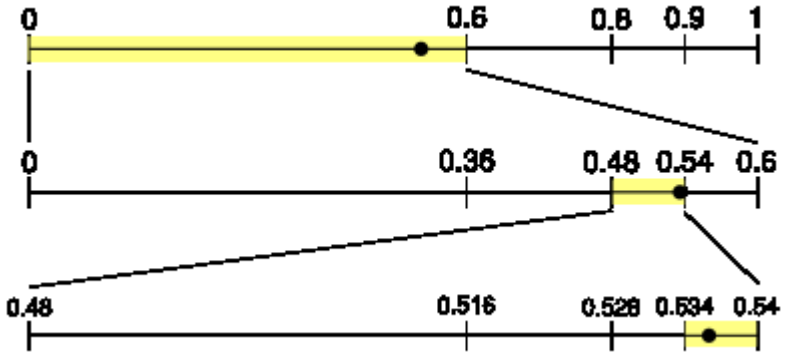
Na výstup dáme číslo z intervalu I napr. 0,538

Pre dekódovanie vstupného čísla v budeme postupovať (obr. 2.19):

1. Nastavíme interval I na <0, 1)
2. Rozdelíme I na podintervaly podľa intervalov jednotlivých znakov a nájdeme znak s s intervalom $\langle D_s, H_s)$, pre ktorý platí $D_s \leq v < H_s$ a I dáme rovné $\langle D_s, H_s)$
3. Vypíšeme znak s a pokračujeme krokom 2

Problém je, že nevieme kedy skončiť. Interval môžeme neustále zmenšovať a vždy sa nájde jeden, do ktorého bude v patriť. Preto je potrebné pridať informáciu navyše. Jednou z možností je pridať počet znakov, ktorý máme očakávať alebo sa pridá nový unikátny znak $, ktorý bude signalizovať koniec reťazca.

Znaku $ už na začiatku priradíme posledný podinterval.



obrázok 2.19: postup dekódovania – prevzaté z [5]

Pri zakódovaní sa krok 4. zmení:

4. Zakódujeme znak $ a na výstup dáme číslo s najkratším desatinným rozvojom z intervalu I

Pri dekódovaní sa krok 3. zmení:

3. Ak je $s = \$$ skončíme inak vypíšeme s a pokračujeme krokom 2

Popísanú metódu robí nepoužiteľnou fakt, že hodnota R sa neustále zmenšuje a pre dlhšie reťazce dostaneme tak malé desatinné čísla, ktoré sa už nedajú v bežných dátových typoch reprezentovať.

Nevýhody vyplývajú z použitia čísel s pohyblivou desatinnou čiarkou. Zaokrúhľovanie spôsobuje malé, ale časom sa kumulujúce chyby, ktoré spôsobia zlyhanie celej metódy. Nevýhodou sú aj pomalšie operácie ako pri číslach s pevnou desatinnou čiarkou.

Ďalšou nevýhodou je potreba prejsť vstupný reťazec dva krát. Najprv pre zistenie počtov výskytov jednotlivých znakov a potom pre samotné zakódovanie. Zistené počty výskytov je nutné v nezakódovanej podobe pridať do archívu, aby sme pri dekódovaní použili rovnaké.

Všetky spomenuté problémy rieši celo číselné adaptívne aritmetické kódovanie.

Kapitola 3

Princípy fungovania programu

3.1 Analýza problému

Program na vstupe dostane zoznam súborov alebo adresárov, ktoré chce užívateľ skomprimovať a mal by vrátiť jeden súbor (archív). Očakávame, že program bude schopný z archívu obnoviť nie len samotné údaje obsiahnuté v súboroch, ale aj celú adresárovú štruktúru, spolu s prístupovými právami prípadne i s vlastníkom a skupinou, do ktorej patrí.

Aby bolo možné adresárovú štruktúru obnoviť na jeden priechod archívu, alebo z prúdu údajov a nebolo potrebné ich uchovávať, je potrebné, aby pri obnove súboru alebo adresára už existoval adresár, do ktorého má byť vytvorený.

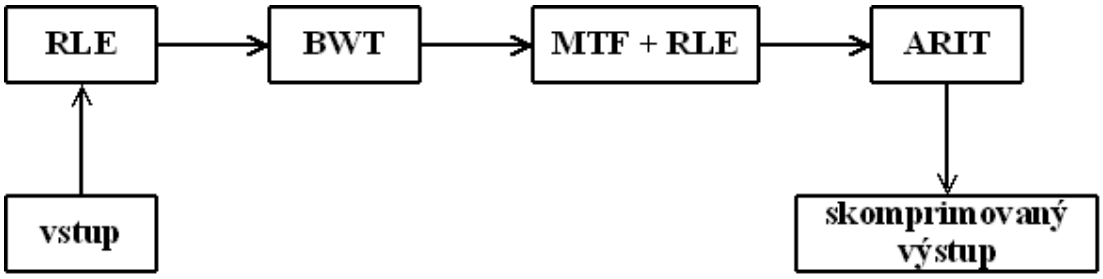
Pri odoberaní z archívu sa presúvaniu súborov v ňom asi nevyhneme, ale pri vhodne zvolenej štruktúre by pridávanie mohlo fungovať len ako prilepenie nových údajov na koniec archívu.

Zo skomprimovaných údajov nie sme schopný zistiť žiadnu dôležitú informáciu o údajoch, ktoré sú skomprimované. Teda keď skomprimujeme spolu dva súbory, nie sme schopný povedať, kde jeden končí a druhý začína. Aj keby sme to vedeli tak získať druhý súbor bez dekompresie celého súboru by nebolo možné. Pri spracovaní celého archívu naraz to nie je problém, ale pri editovaní áno. Sú možné dva prístupy ako to riešiť. Buď sa archív komprimuje a dekomprimuje ako celok, alebo po jednotlivých súboroch. V druhom prípade je potrebné pridať ľahko dostupnú informáciu o tom, čo ktorý skomprimovaný kus obsahuje. A v prvom je potrebné pri každej zmene celý archív dekomprimovať do editovateľnej podoby spraviť zmenu a opäť skomprimovať. Nič nie je ideálne a aj druhé riešenie má nevýhodu. Je ňou kompresný pomer, ktorý jasne hovorí pre prvé riešenie.

3.2 Popis fungovania programu

Hlavná časť programu, ktorá zabezpečuje kompresiu funguje ako spracovateľská linka. (obr. 3.1) Ďalej ju budeme označovať ako KOM. Na vstupe očakáva prúd údajov, ktoré sú najprv spracované metódou RLE, čím sa využijú dlhé behy rovnakých znakov, ktoré sa v údajoch nachádzajú. Po nej nasleduje Burrows-Wheelerova transformácia. Tá má za úlohu preusporiadaním nové behy vytvoriť. Ďalšou v poradí je opäť RLE s rovnakou úlohou, zapísať behy rovnakých znakov do

úspornejšej podoby. Ešte predtým sú údaje spracované metódou MTF, ktorá zvyšuje pravdepodobnosť výskytu znakov s hodnotou blízkou nule, čo zvýši účinnosť poslednej časti a to aritmetického kódovania.

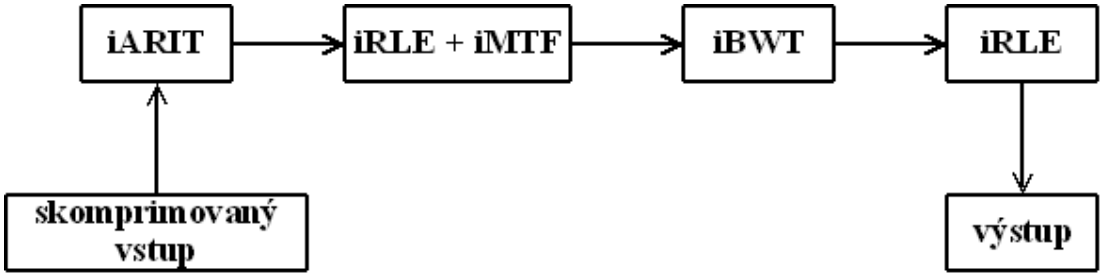


obrázok 3.1: hlavná časť programu zabezpečujúca kompresiu

Každá z častí je nezávislá na ostatných a teda môže a aj beží v samostatnom vlákne. To umožňuje beh na viacerých procesoroch naraz. Výkonovo aj pamäťovo najnáročnejšia časť je BWT a po nej aritmetické kódovanie. ostatné dve vlákna sú v porovnaní s nimi zanedbateľné. Možnosť behu týchto dvoch častí na rôznych procesoroch pozitívne ovplyvní výkon.

Vlákna, ktoré majú spolu komunikovať, dostanú ako parametre smerníky na zdielanú štruktúru TPipe. Tá obsahuje smerník na posielané údaje, ich veľkosť, zámok a conditional variable. Pri každom zápise alebo čítaní z TPipe musí vlákno najprv zamknúť zámok. Ak chce poslať nové údaje a staré ešte neboli prečítané alebo chce čítať a žiadne údaje sa tam nenachádzajú, začne čakať na condtional variable. Druhé vlákno zamkne zámok, zmení stav TPipe, zobudí vlákno, ktoré na conditional variable čakalo a odomkne zámok. Prvé vlákno sa zobudí a ak už sú údaje v očakávanom stave pokračuje ďalej.

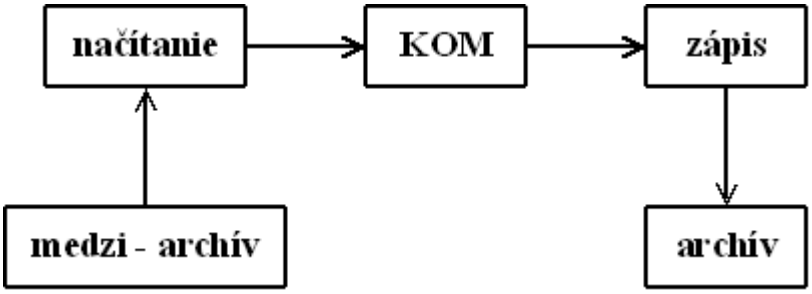
Dekompresia prebieha presne v opačnom poradí s inverznými metódami. (obr. 3.2) Budeme ju označovať DEKOM.



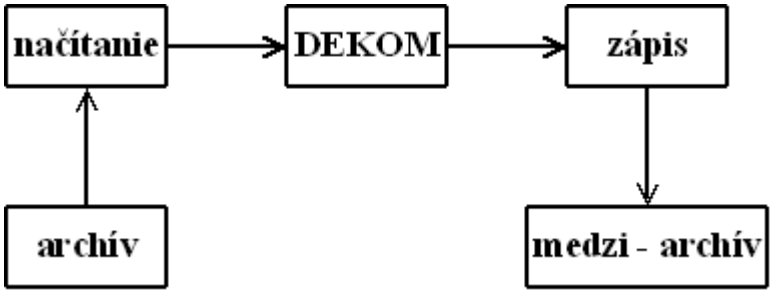
obrázok 3.2: hlavná časť programu zabezpečujúca dekompresiu

BWT je metóda spracúvajúca bloky o veľkosti okolo 1MB a teda kompresie po jednotlivých súboroch by bola neefektívna. Krátke vstupné údaje robia problémy aj aritmetickému kódovaniu, ktorému chvíľu trvá kým správne nainicializuje tabuľku s frekvenciou výskytov. Z toho dôvodu je práca s archívom rozdelená do troch krokov, kde v prvom je vytvorený nekomprimovaný medzi-archív, verne zobrazujúci adresárovú štruktúru a je možné ho editovať. V druhom kroku je skomprimovaný. Pre editovanie je teda potrebné najprv celý archív dekomprimovať (obr. 3.4) a po

uskutočnení zmien v medzi archíve zase skomprimovať (obr. 3.3) ale výhodou je zas lepší kompresný pomer.

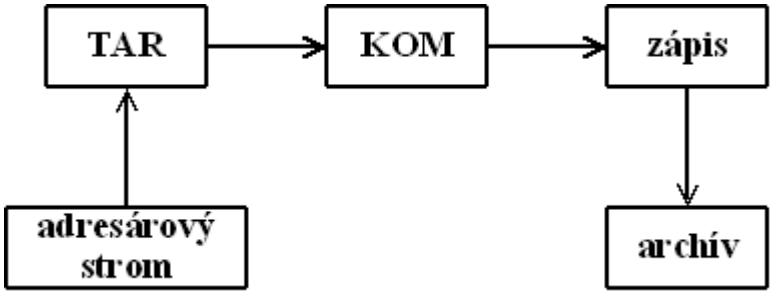


obrázok 3.3: kompresia medzi-archívu

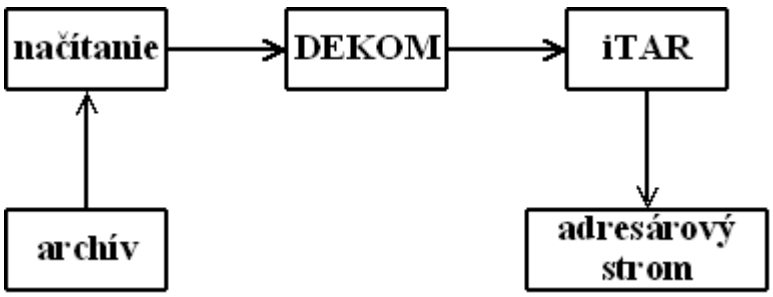


obrázok 3.4: dekompresia archívu do medzi-archívu

Pre metódu spracúvajúcu adresárový strom do podoby jedného súboru budeme používať označenie TAR a metódu k nej inverznú iTAR. Taktiež bežia v samostatných vláknach. Ani nie z dôvodu optimalizácie výkonu ako skôr vytvorenia rovnakého rozhrania pre KOM a DEKOM. (obr. 3.5) (obr. 3.6)



obrázok 3.5: kompresia adresárového stromu



obrázok 3.6: dekompresia do podoby adresárového stromu

3.3 Popis medzi-archívu

Na medzi archív sa môžeme pozerať ako na adresárový strom zabalený do jedného súboru. Pre každý objekt adresárového stromu obsahuje hlavičku so štruktúrou popísanou na obrázku 3.7.

256	cesta k súboru
4	mód súboru
4	ID užívateľa
4	ID skupiny
8	veľkosť súboru
8	dátum poslednej zmeny
1	verzia medzi archívu
1	typ objektu
128	pre cieľ symbolickej linky

obrázok 3.7: štruktúra hlavičky objektu, veľkosti v bytoch

Posledné pole by bolo využité len pre symbolické linky a preto v prípade ostatných objektov sa tam ukladá pokračovanie názvu objektu alebo v prípade blokových a znakových zariadení sa tam ukladá ID zariadenia (rdev)

0 – obyčajný súbor
2 – symbolická linka
3 – znakové zariadenie
4 – blokové zariadenie
5 – adresár
6 – rúra

obrázok 3.8: typy objektov s priradeným číslom

Za hlavičkou nasleduje telo objektu, ktoré má dĺžku zapísanú v hlavičke, ale to len v prípade obyčajných súborov. Ostatné objekty telo nemajú.

3.4 Vytvorenie medzi-archívu

Zabalenie začne prečítaním adresárového stromu z disku a vytvorením troch zoznamov ciest. V prvom sú adresáre v druhom špeciálne objekty a v treťom obyčajné súbory. Adresáre sú do medzi-archívu uložené ako prvé a utriedené podľa cesty. Tým sa zaručí, že pri vytváraní adresáru alebo súboru už bude existovať adresár, v ktorom má byť umiestnený. To umožní rozbalenie len na jeden prechod medzi-archívom. Zoznam obyčajných súborov je utriedený podľa prípon súborov. Tým sa súbory rovnakých typov dostanú blízko seba a vzniknú dlhšie bloky rovnakého typu dát. To umožní aritmetickému kódovaniu lepšie využiť počty výskytov z frekvenčnej tabuľky.

Pri samotnom zabalení sa prechádza zoznam, získavajú sa informácie o objektoch z disku, tie sa prevedú do podoby hlavičky, prípadne sa číta aj obsah súboru. Celé sa to posiela na spracovanie ďalšej časti programu. (obr. 3.5)

Pri čítaní adresárového stromu by sa mohlo stať, že v čase medzi prečítaním obsahu adresára a samotným preskúmaním objektov v ňom niekto nahradí adresár symlinkou. Napr. majme adresár /tmp/hack/etc a užívateľ s právami na čítanie /etc dá zabaliť /tmp.

1. prečíta sa obsah /tmp, pridá sa do zoznamu
2. prečíta sa obsah /tmp/hack, pridá sa do zoznamu
3. útočník nahradí adresár /tmp/hack za symlinku do /
4. pri čítaní obsahu adresára /tmp/hack/etc sa v skutočnosti číta /etc
5. do archívu sa dostanú súbory, ktoré tam nemali byť a čo je horšie po rozbalení bude v adresári /tmp/hach/etc kopia /etc

Podobný problém môže nastať aj pri mazaní celého adresárového stromu. Kde následky môžu byť katastrofálnejšie. Je to spôsobené tým, že funkcie na prácu so súbormi nasledujú symlinky v ceste. Tento problém je v programe odstránený postupným otváraním cesty po jednotlivých adresároch a kontrolou či sa nejedná o symlinku. Teda program zahlási chybu v kroku 4.

3.5 Editovanie medzi-archívu

Pre prácu s medzi-archívom je implementovaných niekoľko funkcií.

- získanie obsahu adresára
- vytvorenie nového adresára
- vykopírovanie časti medzi-archívu na disk
- zmazanie súborov z medzi-archívu
- pridanie nových súborov

Obsah adresára sa zistí na jeden prechod medzi-archívom. Hlavičky sa čítajú za sebou, prípadne sa preskočí telo, ak súbor nejaké má. Z hlavičky sa zistí cesta a tá sa porovná s požadovaným adresárom. Ak sú zhodné až na poslednú časť cesty, ktorú má cesta z archívu navyše, súbor patrí do adresára. Túto funkciu využíva aj funkcia na zmenu adresára, ktorá si takýmto spôsobom overí či adresár vôbec existuje a hneď získa aj jeho obsah. Funkcia na dopĺňanie cesty zadávanej do dialógového okna pracuje tiež na podobnom princípe. Akurát v poslednej časti cesty ešte overí zhodu s prefixom.

Vytvorenie nového adresára by mohlo a aj funguje len ako pridanie informácií o novom adresári na koniec medzi-archívu. Predtým je treba overiť či sa v archíve už adresár s rovnakým menom nenachádza, na čo je potrebný jeden prechod medzi-archívom.

Funkcia na vykopírovanie časti medzi-archívu na disk dostane ako vstup zoznam objektov, ktoré majú byť vykopírované. Pri čítaní hlavičiek sa porovnáva cesta z

menami s medzi-archívu. Pri zhode je objekt vytvorený na disku. V prípade súboru je vykopírované aj jeho telo. Ak je užívateľom, ktorý program spustil, užívateľ root je objektom zmenený vlastník aj skupina podľa záznamu v archíve. Funkcia na vykopírovanie sa používa pri čítaní súboru z medzi-archívu do dočasného súboru, ktorý je po skončení čítania zmazaný.

Zmazanie súborov z medzi-archívu očakáva na vstupe zoznam objektov, ktoré majú byť zmazané. Zhoda nastane, ak cesta zo zoznamu je prefixom cesty z hlavičky. Tým sa zaručí, že v prípade adresárov je zmazaný celý ich podstrom. Všetky objekty v medzi-archíve sú posúvané dopredu o hodnotu premennej, ktorá sa vždy pri zhode ciest zväčší o veľkosť hlavičky a tela objektu.

Pridanie nových súborov je z týchto operácií najzložitejšie. Na vstupe dostane zoznam pridávaných objektov. Tie sa použijú ako korene adresárového stromu, z ktorého sa získajú zoznamy adresárov, špeciálnych súborov a obyčajných súborov. Jedným prechodom sa zo zoznamu adresárov odstránia existujúce adresáre a z medzi-archívu sa odstránia súbory, ktoré sa nachádzali v medzi-archíve aj v zozname. Výsledkom je, že adresáre nebudú zdvojené a súbory budú nahradené. Nakoniec sú objekty zo zoznamov pridané na koniec medzi-archívu.

Kapitola 4

Použité algoritmy

4.1 Larsson-Sadakane

Larsson-Sadakane [2] je algoritmus sufixového triedenia. Popíšeme ho ako je implementovaný v programe. Implementácia bola prevzatá priamo od autorov algoritmu.

Algoritmus na vstup dostane blok dát $X = x_0 x_1 ... x_{n-1}$ tvorený n znakmi a na jeho koniec doplní nový unikátny znak \$ označujúci koniec bloku a platí $\$ < x_i$ pre $0 \leq i < n$.

Sufix začínajúci na i-tej pozícii v bloku X označme $S_i = x_i x_{x+1} ... x_n$ a teda $S_0 = X$ a $S_n = \$$.

Potrebu vytvárania všetkých sufixov (všetkých rotácií) odstránime použitím poľa I o veľkosti n, v ktorom je každý sufix S_i reprezentovaný pomocou i, čo je index na pozíciu v X kde sufix začína. Výstupom algoritmu je pole I usporiadané tak, aby platilo $S_{I[j]} < S_{I[j+1]}$ pre $0 \leq j < n$

Pri použití normálneho porovnávacieho triedenia, by sme najprv utriedili sufixy podľa prvého znaku potom podľa druhého a takto postupovali ďalej. Utriediť by sa nám to podarilo po n krokoch. Usporiadanie, ktoré dostaneme po utriedení podľa prvých k znakov označme usporiadaním U_k. Ak je $k < n$, máme len čiastočné usporiadanie, pretože sa môžu vyskytnúť dva sufixy so spoločným prefixom dlhým aspoň k. Pre zrýchlenie algoritmu využijeme, že netriedime obyčajné nijako nesúvisiace reťazce ale sufixy, pre ktoré platí, že každý sufix je sufixom iného sufixu. Vieme určiť pozíciu pre S_i v usporiadaní U_k, ale tak isto ju vieme určiť aj pre sufix S_{i+k}. Zoradením sufixov najprv podľa pozície S_i a následne dousporiadaním podľa pozície S_{i+k} v usporiadaní U_k dostávame usporiadanie U_{2k}. Tým sa nám podarí znížiť počet krokov potrebných na utriedenie sufixov z n na $\log n$

Nech I je v U_k usporiadaní. Najdlhšia postupnosť sufixov v I, ktoré majú spoločný prefix aspoň dĺžky k bude tvoriť skupinu. V skupine, v ktorej je iba jeden sufix je utriedená. Neutriedená skupina obsahuje aspoň dva sufixy. Najdlhšia postupnosť susedných utriedených skupín je kombinovaná utriedená skupina.

Číslom skupiny, ktorá sa nachádza na $I[f, g]$ bude g.

V poli V bude pre každý sufix uložené číslo skupiny, do ktorej patrí. $V[i]=g$

V poli L budú uložené dĺžky neutriedených a záporné dĺžky kombinovaných utriedených skupín. Teda pre skupinu nachádzajúcu sa na $I[f, g]$ bude v $L[f]$ uložené $g-f+1$ pre neutriedenú a $-(g-f+1)$ pre kombinovanú utriedenú skupinu

1. do poľa I dáme indexy sufixov utriedené podla U_1 a do k priradíme 1 (usporiadanie len podľa prvého znaku)
2. pre každé $0 \le i \le n$ uložíme do $V[i]$ číslo skupiny (pozíciu posledného sufixu s rovnakým počiatočným znakom)
3. pre každú skupinu v $I[f, g]$ nastavíme $L[f]$ na kladnú alebo zápornú dĺžku
4. utriedime každú neutriedenú skupinu v I podľa hodnoty $V[i+k]$ pre $0 \le i \le n$
5. nájdeme miesta kde sa budú neutriedené skupiny deliť (medzi nerovnakými hodnotami)
6. zdvojnásobíme k a vytvoríme nové skupiny rozdelením podľa označených miest, upravíme V a L
7. ak všetky sufixy patria do jednej skupiny, skončíme, inak pokračujeme v bode 4

obrázok 4.1: kroky algoritmu Larsson-Sadakane

	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k	X[i]	B	A	K	A	L	A	R	S	K	A	_	P	R	A	C	A	$
	I[i]	16	1	3	5	9	13	15	0	14	2	8	4	11	6	12	7	10
	V[I[i]]	0	6						7	8	10		11	12	14		15	16
	L[i]	-1	6						-2		2		-2		2		-2	
1	V[I[i] + k]		10	11	14	16	8	0			6	6			15	6		
	I[i]	16	15	13	1	3	5	9	0	14	2	8	4	11	12	6	7	10
	V[I[i]]	0	1	2	3	4	5	6	7	8	10		11	12	13	14	15	16
	L[i]	-9									2		-6					
2	V[I[i] + k]										11	12						
	I[i]	16	15	13	1	3	5	9	0	14	2	8	4	11	12	6	7	10
	V[I[i]]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	L[i]	-17																
	X[I[i]]	$	A	A	A	A	A	A	B	C	K	K	L	P	R	R	S	_
	X[I[i – 1]]	A	C	R	B	K	L	K	$	A	A	S	A	_	P	A	R	A

tabuľka 4.1: postup algoritmu Larsson-Sadakane

Pole L využívame pre kombinované utriedené skupiny na ich preskočenie v konštantnom čase a na určenie koncového bodu pre neutriedené skupiny keď na ne púšťame ďalší krok algoritmu. Koncový bod pre neutriedené skupiny je rovný jej číslu a teda to vieme zistiť priamo z V. Pre uloženie dĺžok kombinovaných utriedených skupín sa nájde miesto v I. Keď sa sufix S_i dostane do kombinovanej utriedenej skupiny, jeho pozícia v I je už konečná a pre ďalší beh algoritmu

nepodstatná. Podstatná je len pre nás, pretože je to výstupná hodnota. Keď sa ale bližšie pozrieme čo po skončení algoritmu obsahuje V, zistíme, že čísla utriedených (jednoprvkových) skupín tvoria inverznú permutáciu k nami hľadanej. Teda sa I dá jednoducho obnoviť z V. (4.1)

$$I[V[i]]=i \text{ pre } 0\leq i\leq n \tag{4.1}$$

Počas behu algoritmu môžeme priestor zabraný už utriedenými skupinami využiť na uloženie zápornej dĺžky kombinovanej utriedenej skupiny. Pre I by platilo, že ak by pri prechode z ľavá do práva bola na začiatku skupiny kladná hodnota, išlo by o neutriedenú skupinu $I[i, V[I[i]]]$, ak záporná, išlo by o kombinovanú utriedenú skupinu $I[i, i-I[i]-1]$.

Po skončení algoritmu sa v I nachádzajú utriedené indexy sufixov, teda pozícia znakov, ktorými sufixy začínajú. My však potrebujeme znaky, ktoré sú na konci rotácií (prechádzajú sufixom). (obr. 2.13) To dosiahneme jednoduchým zmenšením indexov o jedna. Pre výstup Y platí vzorec 4.2.

$$Y[i]=X[I[i]-1] \text{ kde } 0\leq i<n \tag{4.2}$$

Krok 1. v obrázku 4.1 má časovú zložitosť $O(n)$ až $O(n\log(n))$ podľa použitého triediaceho algoritmu. Kroky 2. a 3. len prejdú polia V a L, čo sa dá dosiahnuť so zložitosťou $O(n)$. Kroky 4. až sedem sa môžu opakovať až $\log(n)$ krát. Z nich má najvyššiu časovú zložitosť krok 4., v ktorom dochádza k triedeniu. To zvyčajne beží v čase $O(n\log(n))$ a zdá, že výsledná zložitosť je $O(n(\log(n))^2)$. Hlbším skúmaním sa zistí [2], že je to len $O(n\log(n))$. Pamäťová zložitosť závisí od potrebnej veľkosti indexu do X. Pri nami požadovaných veľkostiach blokov okolo 1MB, 16b index nie je postačujúci a ďalší možný je 32b (8B). Sú používané dve polia I a V o veľkosti n. Výsledná zložitosť je 8n bytov

4.2 Adaptívne celo číselné aritmetické kódovanie

Adaptívne znamená, že pravdepodobnosti a teda aj intervaly pre jednotlivé znaky sa počas behu prispôsobujú priebežne zisteným počtom výskytov. Máme tabuľku, v ktorej si budeme pamätať počty pre doteraz prečítané znaky. Na začiatku je potrebné túto tabuľku inicializovať jednotkami, ako keby sa znak vrátane $ už raz vyskytol. Pri každom prečítanom znaku upravíme počty výskytov v tabuľke.

Máme množinu znakov $A=\{1,2,3,...,n\}$, reťazec $X=x_0x_1...x_k$. Označme I aktuálny interval (4.3) a b je veľkosť, v bitoch, dátového typu, ktorý použijeme na uloženie D a R.

$$I=\langle D, D+R)\tag{4.3}$$

Na začiatku inicializujeme $D=0$ a $R=2^{b-1}$. Počet výskytov znaku i pre $0<i\leq n$ označíme c_i a jeho kumulatívny počet výskytov m_i. (4.4) Preň platí $m_0=0$ a $m=m_n$.

$$m_i=\sum_{j=1}^{i} c_j \tag{4.4}$$

Interval K_s (4.5) je interval prislúchajúci znaku s a je určený počtom výskytov. (4.6) (4.7)

$$K_s=\langle D_s, D_s+R_s) \tag{4.5}$$

$$D_s=\frac{m_s}{m} \tag{4.6}$$

$$R_s=\frac{m_s-m_{s-1}}{m} \tag{4.7}$$

Úprava intervalu I sa vypočíta dosadením do vzorca pre čísla s pohyblivou desatinnou čiarkou. (2.1) (2.2)

$$D=D+R*\frac{m_{s-1}}{m} \tag{4.8}$$

$$R=R*\frac{m_s-m_{s-1}}{m} \tag{4.9}$$

Je potrebné zaručiť, že chyba spôsobená celo číselným delením bude čo najmenšia, preto vzorce 4.10 a 4.11.

$$D=D+\left\lfloor\frac{R}{m}\right\rfloor*m_{s-1} \tag{4.10}$$

$$R=\left\lfloor\frac{R}{m}\right\rfloor*(m_s-m_{s-1}) \tag{4.11}$$

Ďalej je potrebné zaručiť, že pomer R a m neprekročí určité hranice. Z toho dôvodu musí R spĺňať invariant (4.12) a malo by platiť $m<2^{b-8}$ [3]. Preto je potrebné pri náraste m nad kritickú hodnotu. Tabuľku znulovať, alebo aspoň počty výskytov zmenšiť, napr. na polovicu.

$$2^{b-2}<R\leq 2^{b-1} \tag{4.12}$$

Ale aj tak vznikne chyba pri každom delení, ktorá posunie niektoré intervaly doľava a skráti ich. Tým na konci R zostáva nepokrytá časť. Ošetrí sa roztiahnutím

posledného intervalu na potrebnú veľkosť (kód 4.1). D a R sú uložené v konečných a pomerne malých dátových typoch oproti vstupným reťazcom, ktorých interval by mali zvládnuť reprezentovať. Hodnota R sa každým načítaným znakom zmenšuje a treba robiť kroky na udržanie hodnoty v požadovaných medziach. (4.12)

```
if ( s = n )
      R = R - R / m * m_{n-1}
else
      R = R / m * (m_s - m_{s-1})
```
kód 4.1: ošetrenie natiahnutia posledného intervalu

Keď je $R<2^{b-2}$, nespĺňa invariant (4.12) a je potrebné ho zväčšiť. Vtedy sa skontroluje či interval I padne celý do $\langle 0, 2^{b-1})$. Ak áno binárny zápis jeho hornej aj dolnej medze začína nulou. Keďže nové hodnoty intervalu I, vypočítané podľa vzorca 4.10 a 4.11, sú vždy podintervalom pôvodného, tá nula na začiatku sa už nezmení. Teda ju môžeme vypísať na výstup a hodnoty D a R prenásobiť dvoma. Podobný prípad nastane keď I padne celý do $\langle 2^{b-1}, 2^{b})$, akurát s jednotkou na začiatku. Tu sa D ešte pred prenásobením zmenší o 2^{b-1} hodnota, aby nepretieklo nad 2^{b}.

Môže sa stať, že $R<2^{b-2}$, ale nenastane ani jeden z popísaných prípadov. Časť I padne do $\langle 0, 2^{b-1})$ a druhá $\langle 2^{b-1}, 2^{b})$. V binárnom zápise bude dolná medza začínať 01 a horná 10. Nie je jasné, aká hodnota sa môže dať na výstup. Do k sa bude značiť koľko krát daný stav nastal. Pred prenásobením dvoma sa hodnota D zmenší o 2^{b-2}. Keď neskôr I padne celé do dolnej alebo hornej časti, vypíšeme 0 alebo 1, za ňou k - krát 1 alebo 0 a k znulujeme.

```
POL = 2^{b-1}
STVRT = 2^{b-2}

tabulka.daj_hodnoty(s, m_{s-1}, m_s, m)  //získa potrebné hodnoty pre s
r = R / m
D = D + r *(m_{s-1})
if ( m_s < m )      //iný ako posledný znak
      R = r * (m_s - m_{s-1})
else  //posledný znak
      R = R - r*(m_{s-1})
while ( R <= STVRT ){   //kým nespĺňa invariant
      if ( D >= POL ){  //celý v hornej polovici
            posli_bity(true)  //vypíše 1 a k-krát 0
            D = D - POL
      } else if ( D + R <= POL ){   //celý v dolnej polovici
            posli_bity(false) //vypíše 0 a k-krát 1
      } else {    //ani tam ani tam
            ++k
            D = D - STVRT
      }
      D_z<<=1
      R_z<<=1
}
tabulka.uprav(s)  //zvýši počet výskytov pre s
```
kód 4.2: zakódovanie symbolu s

```
POL = 2^(b-1)
STVRT = 2^(b-2)

m = tabulka.daj_m()      //získa celkový súčet výskytov znakov
r = R / m
ciel = min(m-1, D / r)  //hodnota, pre ktorú treba nájsť interval

tabulka.najdi(ciel, s, m_(s-1), m_s)      //nájde interval pre ciel a
vyplní hodnoty

D = D - r *(m_(s-1))
if ( m_s < m ){ //iný ako posledný znak
      R = r * (m_s - m_(s-1))
} else { //posledný znak
      R = R - r * (m_(s-1))
}
while ( R <= STVRT ){   //kým nespĺňa invariant
      R = (R << 1)
      if ( daj_bit() )  //načíta vstup
            D = (D<<1) | 1
      else
            D = (D_r<<1)
}
return s
```
kód 4.3: dekódovanie symbolu s

$$D = D + \frac{R}{2} \tag{4.13}$$

Po zakódovaní symbolu \$, sa upraví D (4.13) a vypíše na výstup. Nezmenenie hodnoty D, by mohlo spôsobiť nerozpoznanie znaku \$, kvôli zaokrúhleniu.

4.3 Moffatova tabuľka kumulovaných počtov výskytov

Pri aritmetickom kódovaní si potrebujeme pre svoju činnosť pamätať počty výskytov jednotlivých znakov.

Pri zakódovaní jedného znaku potrebujeme:
1. počet výskytov znaku
2. kumulovaný počet výskytov znaku
3. počet výskytov všetkých znakov
4. zvýšiť počet výskytov znaku o 1

Pri dekódovaní jedného znaku potrebujeme:
1. počet výskytov znaku
2. kumulovaný počet výskytov znaku
3. počet výskytov všetkých znakov
4. zvýšiť počet výskytov znaku o 1
5. podľa kumulovaného počtu výskytov nájsť, o ktorý znak ide

Ak by sme tabuľku mali implementovanú len ako pole veľkosti n, kde by na s-tej pozícii bol kumulovaný počet výskytov pre znak s, prvý, druhý a tretí krok by sa dal spraviť so zložitosťou $O(1)$ a štvrtý v $O(n-s)$, pretože treba zväčšiť aj hodnoty pre znaky nasledujúce v poli za znakom s. Piaty krok sa dá pomocou binárneho vyhľadávania spraviť v $O(\log(n))$.

Ak by sme do tabuľky ukladali len počty znakov, štvrtý a prvý krok by sa dal spraviť v $O(1)$, tretí za pomoci premennej,v ktorej by sa udržiaval celkový počet znakov nezávisle na tabuľke, je tiež $O(1)$, ale druhý a piaty krok bežia v $O(s)$ takže k zlepšeniu nedošlo.

Dátová štruktúra, ktorá lepšie vyhovuje požiadavkám aritmetického kódovania je Moffatova tabuľka kumulovaných počtov výskytov [1] a to je aj dôvod, prečo je implementovaná v programe.

Označme n počet všetkých znakov, c_i počet výskytov i-tého znaku a s -ty znak chceme zakódovať. Zadefinujme M ako Moffatovu tabuľku.

$l_s = \sum_{i=1}^{s-1} c_i$ - kumulovaný počet výskytov (dolná hranica)

$h_s = \sum_{i=1}^{s} c_i = l_s + c_s$ - kumulovaný počet výskytov (horná hranica)

$m = \sum_{i=1}^{n} c_i$ - celkový kumulovaný počet

$$\text{veľkosť}(s) = max\{2^v ; s \bmod 2^v = 0, v = 0,1,2,...\}$$

$$\text{dopredu}(s) = s + \text{veľkosť}(s) \qquad \text{dozadu}(s) = s - \text{veľkosť}(s)$$

V M nie je na pozícii s uložený počet výskytov ani kumulovaný počet výskytov ale platí $M[s] = \sum_{i=s}^{\text{dopredu}(s)-1} c_i$. (tab. 4.2)

s	1	2	3	4	5	6	7	8	9
c_s	15	10	8	5	5	4	4	2	1
l_s	0	15	25	33	38	43	47	51	53
h_s	15	25	33	38	43	47	51	53	54
dozadu(s)	0	0	2	0	4	4	6	0	8
dopredu(s)	2	4	4	8	6	8	8	16	10
rozsah	1-1	2-3	3-3	4-7	5-5	6-7	7-7	8-9	9-9
M[s]	15	18	8	18	5	8	4	3	1

tabuľka 4.2: prehľad hodnôt premenných, Moffat

Počet výskytov pre s získame tak, že od $M[s]$ odpočítavame hodnoty, ktoré získavame prechádzaním M pomocou dopredu() od $s+1$ do dopredu(s). Časová zložitosť je $O(1)$, keďže dĺžka iterácie je pre $n/2$ hodnôt dlhá 0, pre $n/4$ 1, pre $n/8$ 2, ...

```
c = M[s]
q = s + 1
z = min(dopredu(s), n + 1)
while(q < z) {
      c = c - M[q]
      q = dopredu(q)
}
return c
```
kód 4.4: zistí počet výskytov pre s, Moffat

Príklad: nech $s=4$ potom $z=8$ a q postupne nadobúda hodnoty 5, 6 a 8

Na zistenie kumulovaného počtu výskytov najprv spočítame $M[1]+M[2]+M[4]+...+M[2^{\lfloor \log_2(s) \rfloor -1}]$ čo je viac ako l_s. Potom prechádzame M pomocou dopredu() od s do $M[2^{\lfloor \log_2(s) \rfloor}]$ a odpočítame hodnoty v M. Ako je vidieť prvý krok beží so zložitosťou $O(\log(s))$ a druhý tiež.

```
p = 1
l = 0
while(p < s) {
      l = l + M[p]
      p = 2 * p
}
q = s
while(q != p && q <= n) {
      l = l - M[q]
      q = dopredu(q)
}
return l
```
kód 4.5: zistí kumulovaný počet výskytov pre s, Moffat

Príklad: nech $s=5$ potom p nadobúda hodnoty 1, 2, 4, 8 a q postupne nadobúda hodnoty 5, 6 a 8

Zväčšením hodnôt, na ktoré sa dostaneme prechádzaním M pomocou dozadu() od s po 0, o 1, zväčšíme počet výskytov pre s o 1. Časová zložitosť je $O(\log(s))$.

```
p = s
while(p > 0) {
      M[p] = M[p] + 1
      p = dozadu(p)
}
```
kód: 4.6: zväčší počet výskytov pre s o 1, Moffat

Príklad: nech $s=5$ potom p nadobúda hodnoty 5, 4, 2, 1 a 0

Pri dekódovaní nám Moffatova tabuľka umožňuje spojiť nájdenie znaku podľa cieľa, zväčšenie hodnoty a zistenie kumulatívnych počtov do jedného prechodu M. Počet výskytov je nutné získať samostatnou funkciou. V prvej časti nájdeme $min\{p; M[p] < cieľ, p = 1,2,4,...\}$. V druhej medzi pozíciami, ktoré má $M[p]$ v rozsahu, hľadáme znak s, pre ktorý platí $l_s \leq cieľ < l_s + c_s$.

```
p = 1
l = 0
while((2 * p) <= n && M[p] <= ciel) {
      ciel = ciel - M[p]
      l = l + M[p]
      p = 2 * p
}
s = p
m = p/2
e = 0
while(m >= 1) {
      if((s + m) <= n) {
            e = e + M[s + m]
            if((M[s] - e) <= ciel) {
                  ciel = ciel - (M[s] - e)
                  l = l + (M[s] - e)
                  M[s] = M[s] + 1
                  s = s + m
                  e = 0
            }
      }
      m = m/2
}
M[s] = M[s] + 1
return s, l
```

kód: 4.7: nájde s podľa cieľa, zväčší počet výskytov pre s o 1 a získa hodnoty, Moffat

Príklad: nech $cieľ = 40$ potom p nadobúda hodnoty 1, 2 a 4, e nadobúda hodnoty 8, 13 a 0 a s hodnoty 4 a 5

Moffatova tabuľka kumulovaných počtov výskytov vykoná kroky 1. až 4. ale aj 1. až 5. pre znak s v čase $O(\log(s))$. Aritmetické kódovanie z kapitoly 4.2 teda bude pracovať v čase $O(n + c + m(\log(s)))$ pre zakódovanie aj dekódovanie. n je potrebné na nainicializovanie tabuľky, m je počet spracovávaných znakov a c je výsledná dĺžka kódu.

Kapitola 5

Dosiahnuté parametre programu

5.1 Princípy merania výkonu

Hlavné výkonové parametre kompresných programov sú čas kompresie a pomer veľkostí archívu a pôvodných dát. Keďže dáta môžu byť svojím obsahom veľmi odlišné a programy môžu dosahovať rôzne výsledky aj na rovnako veľkých dátach rôznych typov, je potrebné pre porovnávanie zaručiť rovnaké podmienky. Preto vznikli korpusy. [4] Korpus je výber niekoľkých súborov, ktorý je pevne daný a v čase sa nemí. To zaručuje rovnaké dáta pre testovanie kompresných programov a teda je možné porovnanie kvality kompresie. Pre porovnanie rýchlosti je navyše potrebné zladiť aj počítač. V podstate jedinou možnosťou je spustenie všetkých programov, ktoré majú byť porovnávané, na tom istom počítači na tie isté dáta.

5.2 Porovnanie rýchlosti a efektivity

Program k tejto práci by mal byť približne na úrovni programu bzip2. Takže porovnávať budeme s týmto programom. Ako testovacie dáta bol zvolený výber súborov, na ktorých kompresiu bude program užívateľmi asi najčastejšie používaný. (tab. 5.1) Aby bola zaručená, čo najväčšia presnosť, bola snaha vyberať veľké súbory, na ktorých sa rôzne nepredvídateľné vplyvy stanú zanedbateľnými. Práve korpusy nespĺňali podmienku veľkosti súborov.

Testovacie dáta sú tvorené len súbormi, pretože bzip2 adresáre nepodporuje. Testovanie bolo robené na PC (x86-64), AMD 64 X2 3800+ (2x2GHz) - socket AM2, 2048 MB RAM, Gentoo Linux

Časy komprimácie sú merané v sekundách, čím je meranie, najmä pre menšie súbory, pomerne nepresné. To však nie je v protiklade so zamýšľaným účelom merania a to poskytnúť len informatívne výsledky.

1	file.odt	textový dokument
2	file.exe	spustiteľný súbor, ale nie inštalačný
3	file.html	čisté html
4	file.mp3	mp3
5	file.pdf	pdf
6	file.mtr	pakovatkom vytvorený archiv
7	installed_prog.tar	tar adresára nainštalovaného programu
8	java_source.tar	tar adresára Java projektu
9	user_data.tar	tar adresára s uživatelským dátami: obrázky, dokumenty ...

tabuľka 5.1: súbory pre testovanie výkonu

	veľkosť (B)			**kompresný pomer**		**čas (s)**	
	pôvodne	**bzip2**	**pakovatko**	**bzip2**	**pakovatko**	**bzip2**	**pakovatko**
1	5812736	1594632	1826051	27,43%	31,41%	3	4
2	4259892	2036915	2163617	47,82%	50,79%	1	3
3	12511614	2484496	2562002	19,86%	20,48%	4	7
4	4391609	4375429	4398375	99,63%	100,15%	2	3
5	6585525	6413225	6414168	97,38%	97,40%	3	5
6	3853971	3871040	3881235	100,44%	100,71%	2	3
7	26229248	7488542	7821291	28,55%	29,82%	8	12
8	21384192	4063536	3853972	19,00%	18,02%	7	11
9	82400768	51431373	56726767	62,42%	68,84%	38	37

tabuľka 5.2: výsledky testovania výkonu

Kapitola 6

Použitie

6.1 Inštalácia

Program pakovatko je programovaný v jazyku C++ a je určený pre UNIX-ovú konzolu. Vyvíjaný a prevažne testovaný bol na Linux-e. Úspešne prebehli testy aj na FreeBSD a Solaris-e, podpora na ostatných systémoch nie je zaručená.

Požadované programové vybavenie:

- gcc 3.4.1 a vyššie
- knižnica ncurses, testované s verziou 5.6

Pri testovaní boli zaznamenané problémy práve s knižnicou ncurses. Predinštalovaná knižnica na systémoch nepracovala správne. Prejavovalo sa to nezobrazovaním niektorých častí užívateľského prostredia, čo značne obmedzovalo použiteľnosť. Je možné, že to bolo spôsobené zlou inštaláciou. Po stiahnutí z [6], skompilovaní a nainštalovaní ncurses verzie 5.6 sa problémy odstránili.

Program je dostupný v podobe súborov so zdrojovým kódom. Do spustiteľnej podoby sa dostane štandardnou postupnosťou príkazov, spustených z hlavného adresára:

- ./configure – zistí aktuálne parametre systému
- make – vytvorí spustiteľný súbor

Ak prebehli v poriadku, program je pripravený na použite. Spustí sa príkazom ./src/pakovatko.

Príkaz make install skopíruje spustiteľný súbor programu na miesto, ktoré by malo byť dostupné všetkým užívateľom, štandardne /usr/local/bin/pakovatko.

V prípade potreby je možné skriptom ./autogen, ktorý využíva „GNU automake“ utilitu, vygenerovať nové configure špeciálne pre konkrétny systém.

6.2 Užívateľské prostredie

Užívateľské prostredie programu je tvorené dvoma oknami. Ľavé poskytuje prístup k adresárovému stromu na disku a pravé, v prípade otvorenia archívu, k adresárovému stromu uloženému v archíve. Obe okná sú identické až na funkčnosť niekoľko málo

kláves. V hornej časti každého z nich je vypísaná cesta k aktuálnemu adresáru a v okne je zobrazený jeho obsah. V pravom okne sa za menom aktuálneho adresára zobrazuje znak '*', čo symbolizuje, že zmena archívu ešte nebola do archívu skomprimovaná. Program zmeny archívu ukladá okamžite, takže k zobrazeniu '*' môže dôjsť len pri zlyhaní ukladania zmien.

Spoločné klávesy pre obe okná:

- F1 / 1 / ? - sú najdôležitejšie klávesy, pretože zobrazia okno s nápovedou
- šípka hore / k – posunie kurzor o položku vyššie
- šípka dole / m – posunie kurzor o položku nižšie
- home / ctrl + A – presunie kurzor na prvú položku v adresári
- end / ctrl + E – presunie kurzor na poslednú položku v adresári
- page up / ctrl + B - presunie kurzor o stranu vyššie
- page dn / ctrl + D - presunie kurzor o stranu nižšie
- enter / o – v prípade ak je aktuálna položka adresár alebo symbolická linka na adresár, zmení naň aktuálny adresár
    - v prípade ľavého okna a obyčajného súboru alebo symbolickej linky na obyčajný súbor sa program pokúsi otvoriť tento súbor ako archív a jeho adresárový strom načíta do pravého okna. Ak súbor nespĺňa požadovaný formát program zahlási chybu.
- insert / space – vyznačí alebo odznačí aktuálnu položku, v dolnej časti okna sa zobrazuje počet a celková veľkosť vyznačených položiek
- back space – zmení aktuálny adresár na rodičovský adresár
- tab / t – prepne ovládanie do druhého okna
- e – zobrazí posledný zoznam chýb
- r / ctrl + L – prekreslí a upraví veľkosť okien, opraví terminál
- F2 / 2 – otvorí dialógové okno, do ktorého sa zadá cesta, na ktorú sa požaduje zmeniť aktuálny adresár
- F3 / 3 – otvorí pre čítanie aktuálnu položku, ak je typu obyčajný súbor alebo symbolická linka na obyčajný súbor
- F5 / 5 – je možné použiť len v prípade, ak je otvorený nejaký archív. Má rozdielnu funkčnosť pre ľavé a pravé okno.
    - ľavé okno – vyznačené položky, alebo aktuálnu, ak žiadne neboli označené, pridá do archívu. V prípade, že sa už v archíve nachádzajú budú nahradené.
    - pravé okno – vyznačené položky, alebo aktuálnu, ak žiadne neboli označené, vykopíruje z archívu na disk. Prepísatelné súbory budú prepísané.
- F7 / 7 – vytvorí nový adresár podľa mena zadaného do dialógového okna
- F8 / 8 – zmaže označené položky alebo aktuálnu, ak žiadne neboli označené, predtým sa ešte spýta na potvrdenie operácie
- F10 / q / ESC – ukončí program

Klávesy funkčné len v ľavom okne:

- F4 / 4 – otvorí pre editovanie aktuálnu položku, ak je typu obyčajný súbor alebo symbolická linka na obyčajný súbor
- F11 / - – otvorí dialógové okno pre skomprimovanie vyznačených, alebo aktuálnej položky. Do okna je potrebné zadať cieľový súbor a prípadne

zmeniť veľkosť bloku. Pole s veľkosťou bloku akceptuje len číslice a veľkosť bloku by mala byť v rozmedzí 100 až 2500. Tento parameter ovplyvňuje rýchlosť a kvalitu kompresie. Čím väčší blok, tým by mala byť komprimácia pomalšia ale lepšia.

- F12 / = – otvorí sa dialógové okno, do ktorého sa zadá cieľový adresár. Program sa pokúsi aktuálny súbor otvoriť ako archív a dekomprimovať ho do zadaného adresára

Klávesy funkčné len v ľavom okne:

- c – ak je v pravom okne otvorený archív, zavrie ho
- p – ak je v pravom okne otvorený archív, skomprimuje ho a uloží na jeho pôvodné miesto. Tým sa do archívu skomprimujú všetky zmeny, ktoré boli vykonané.

Funkcie kláves v dialógoch:

- tab – v poliach kde sa zadáva cesta k súboru alebo adresáru, zobrazí zoznam doplniteľných mien, súborov alebo adresárov
- F10 / q / ESC – ukončí dialóg

Progress bar reaguje iba na klávesy q a ctrl + C, ktoré spôsobia zastavenie aktuálne prebiehajúcej operácie

V prípade, že dôjde k neočakávanému ukončeniu programu (výpadok elektriny, porucha počítača) a zmeny spravené v pravom okne nestihli byť skomprimované do archívu, je možné program spustiť s parametrami „cesta k archívu“ „cesta k dočasnému súboru“ „veľkosť bloku“. Temp súbor je zvyčajne uložený v /tmp a jeho názov má tvar pakovatkoXXXXXX. Je potrebné, aby temp súbor mal očakávaný formát. Ten si program overí hneď po otvorení a skomprimuje temp súbor do archívu zadaného parametrom.

Kapitola 7

Záver

7.1 Záver

Algoritmy a postupy popísané v úvode sa podarilo implementovať. Užívateľské rozhranie podporuje funkcie stanovené v prvej časti zadania. Nie je možné, aby spĺňalo požiadavky každého užívateľa, ale k tomuto ideálnemu stavu sa snaží čo najviac priblížiť. O splnení druhej časti zadania jasne hovorí tabuľka číslo 5.2. Z nej je vidieť, že navrhnutý a implementovaný kompresný algoritmus je približne na úrovni programu bzip2.

Ďalší vývoj by mohol byť zameraný na zlepšenie algoritmu na kompresiu, tak aby dosahoval lepšie výsledky ako program bzip2. Je pravdepodobné, že by bolo treba navrhnúť úplne nový algoritmus a jeho vytvorenie sa dá považovať za náročnú úlohu.

Program bol testovaný na rôznych typoch dát niekoľkými užívateľmi a v súčasnej podobe by nemal obsahovať žiadnu chybu, ktorá by mohla spôsobiť nekorektné správanie a poškodenie alebo stratu dát.

Literatúra

[1] A. Moffat: An Improved Data Structure for Cumulative Probability Tables. *Software-Practice and Experience*, **29(7)**, 647-659 (1999)

[2] N. Jesper Larsson, Kunihiko Sadakane: *Faster Suffix Sorting,* http://www.larsson.dogma.net/ssrev-tr.pdf

[3] RNDr. Tomáš Dvořák, Csc.: materiály k prednáške Algoritmy komprese dat, http://ksvi.mff.cuni.cz/~dvorak/vyuka/SWI072/Aritmet.pdf

[4] http://links.uwaterloo.ca/calgary.corpus.html

[5] http://www.wikipedia.org

[6] ftp://ftp.gnu.org/gnu/ncurses/